LA-UR -86-4336

CONF-8704171--1

Los Alamos National Laboratory is operated by the University of California for the United States Department of Energy under contract W-7405-ENG-36.

Los Alamos Los Alamos National Laboratory
Los Alamos, New Mexico 87545

TITLE: IMPLEMENTING REMOTE PROCEDURE CALLS WITH DECNET

LA-UR--86-4336

DE87 003754

AUTHOR(S): E. Bjorklund and S. C. Schaller

SUBMITTED TO: DECUS Refereed Papers Competition
DECUS Spring Meeting, Nashville, TN,
April 27-May 1, 1987

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

By acceptance of this article, the publisher recognizes that the U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or to allow others to do so, for U.S. Government purposes.

The Los Alamos National Laboratory requests that the publisher identify this article as work performed under the auspices of the U.S. Department of Energy

MASTER

Los Alamos Los Alamos National Laboratory
Los Alamos, New Mexico 87545

FORM NO 836 R4
ST NO 2629 5/81

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

IMPLEMENTING REMOTE PROCEDURE CALLS WITH DECNET*

By
Eric Bjorklund and Stuart C. Schaller
Los Alamos National Laboratory
Los Alamos, New Mexico 87545

ABSTRACT

The "Remote Procedure Call" (RPC) has recently become an important communication model for distributed systems. The basic idea behind remote procedure calls is that a process running on one machine can "call", using standard procedure calling semantics, another routine that executes on a different machine. A message-passing mechanism is used to transfer parameters between the caller and the called routine.

In this paper, we describe a remote procedure call system we have implemented that uses DECnet as the underlying message-passing mechanism. Our system is highly reliable, reasonably efficient, and supports some advanced features such as asynchronous remote procedures. The described system is currently part of a distributed accelerator-control system containing VMS, Micro-VMS, and VAXELN nodes. It could also be extended to any other system that supports DECnet. Topics discussed include the system design, parameter-passing protocol, error detection and recovery, and performance.

1 INTRODUCTION

The principal advantage of using remote procedure calls (RPC) for inter-process communication is the simplicity of the concept. The procedure call is a well understood mechanism and a useful tool for

---

* Work supported by the US Department of Energy

dealing with abstraction. Remote procedure calls have been implemented in several distributed systems such as the Xerox Cedar project [2] [5] and the University of Washington's Eden project [1]. In addition, any distributed system written in the Ada™ programming language might also be considered an RPC system since the semantics of the Ada "Rendezvous" [4] are very similar to those of a remote procedure call.

1.1 Goals Of The LAMPF RPC System

The RPC system described in this paper was designed to support a distributed control system for the 800-MeV linear accelerator at the Los Alamos Meson Physics Facility (LAMPF). The LAMPF control system uses a VAX 780 as the central control computer, another VAX 780 as a development computer and backup control computer, and several Micro-VAX II computers for various specialized tasks. Most of the computers in the system are connected via Ethernet. The two 780's run VMS. The Micro-VAXs run either Micro-VMS or VAXELN. Our system had to meet the following criteria:

- The system must be easy to use (the whole point of an RPC system was to simplify inter-process communication).
- The system should be easy to implement. Since the system must be implemented on two different operating systems, that task should not be made any more difficult than necessary. Unfortunately, software at the level of an RPC interface does not lend itself well to transportability. The next best thing was to keep the number of lines of code to a minimum.
- The system must support processes that are written in different languages, that run under different operating systems, and communicate over different media. In addition to being able to

communicate across a network, the system should also support communication between processes residing on the same node.

- Speed is not of the essence (but...). Most of the time-critical functions in our control system are performed by dedicated hardware, or by moderately dedicated remote computers. Still, the system should not be so slow that it exceeds the operator frustration threshold.

1.2 Design Decisions

Given the constraints mentioned in the previous section, we made some design decisions that differ from many of the other currently implemented RPC systems. Two of these decisions were: 1) to support both synchronous and asynchronous procedure calls, and 2) to use DECnet rather than design our own communications protocol.

1.3 The Use Of DECnet

Many RPC systems implement their own communications protocol. The reason for this is that the RPC model is simple enough that a specialized communications protocol can be faster and more efficient than a more general message-passing system such as DECnet. This is especially true in a homogeneous system where there is only one communications medium, one processor type, and one operating system. Given the heterogeneity of our network and the desire to keep the RPC interface easy to implement, we decided to let DECnet handle the basic communications problems of message delivery, routing, checksumming, and error retries.

2 THE STRUCTURE OF THE RPC INTERFACE

The RPC interface is divided into two parts, a "Caller's

Interface" and a "Server's Interface". Figure 1 illustrates the structure of the RPC interface for the case where the caller is on a VMS node and the server is on a VAXELN node (one of the more common cases in our control system).

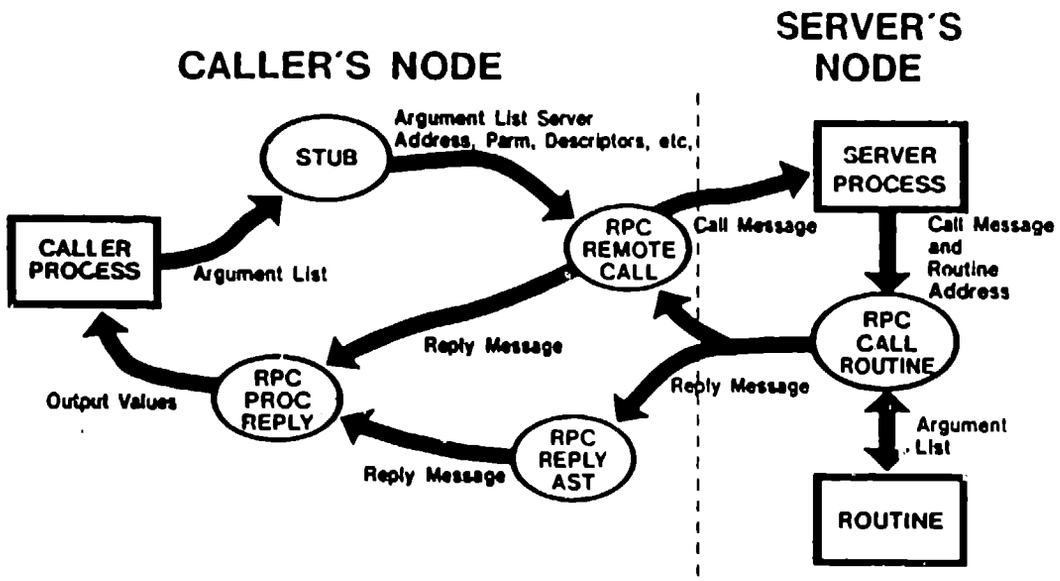

Figure 1
RPC System Structure

2.1 The Caller's Interface

On the caller's node, the calling process is linked to a stub routine which has the same name and the same set of parameters as the remote procedure. The stub routine passes its parameters, along with some additional information, to the RPC interface routine, RPC_REMOTE_CALL. RPC_REMOTE_CALL packs the parameters into a call message and sends the call message over the network to the server process. If the call is synchronous, the calling process is blocked until the remote procedure completes. If the call is asynchronous, control is returned to the caller, and when the remote procedure completes the AST routine (RPC_REPLY_AST) will execute.

When the remote procedure completes, the server's RPC interface sends a "reply message" back to the caller. The reply message is read by RPC_REMOTE_CALL (for synchronous calls) or RPC_REPLY_AST (for asynchronous calls) and is passed to the routine, RPC_PROC_REPLY. RPC_PROC_REPLY unpacks the reply message and writes the values of the output parameters into their variables.

2.2 The Server Process

Procedures are not usually stand-alone entities -- they are generally contained within some environment such as a process or an operating system. In our system, remote procedures are contained within special processes called "server processes". As the name implies, a "server process" provides a service, such as analyzing data or controlling a device. It also provides a set of remote procedures that access this service.

The job of a server process is to listen for call messages, determine which procedure should be invoked, and pass the address of that procedure, along with the call message, to the server's RPC interface routine, RPC_CALL_ROUTINE. RPC_CALL_ROUTINE re-creates the argument list from the call message and then calls the specified procedure. When the procedure returns, RPC_CALL_ROUTINE packs the output parameters into a reply message which it then sends back to the caller process.

3 ASYNCHRONOUS REMOTE PROCEDURE CALLS

Some RPC systems support asynchronous remote procedure calls while other systems only support synchronous calls. Whether or not asynchronous calls are necessary seems to depend on how well the operating system supports local concurrency [3] (allowing a single task to have multiple threads of control). VAXELN provides good

support for local concurrency. We decided, however, that we needed to support asynchronous remote procedure calls on VMS nodes.

One of the problems with asynchronous procedures is knowing when the procedure has completed. There are several methods a program may use to accomplish this. It can poll one of the procedure's output variables to see if it has been written yet, or it can wait on some event or semaphore that is signaled by the procedure. Some systems also allow for a special interrupt routine to be invoked when an asynchronous activity completes.

Our RPC system provides all three of the above methods (depending on the operating system). On a VMS or Micro-VMS node, the caller can poll an output variable, wait for an event flag, schedule an AST, or any combination of these three.

Sometimes the caller does not care when, or even if, the remote procedure completes. In this case, the call can be a "No-Reply" call and the remote procedure will execute independently of the calling process. The server's RPC interface will not send back a reply message when the procedure completes. No-reply calls should therefore not be made to procedures with output or modified parameters.

As might be expected, synchronous calls comprise the majority of the remote procedure calls in our control system. Asynchronous and no-reply calls are generally used only when increased performance is required.

4 PROCEDURE BINDING

Traditional procedures are normally bound to their callers at either compile or link time. Remote procedures, however, must generally be bound at run time. Three things are required to bind a caller to a remote procedure in our system: the name of the server

process, the name of the node the server process is running on, and the name of the procedure to call.

The node and server names are used to create a DECnet "logical link" between the calling process and the server process. The procedure name is translated into a "procedure ID" and sent to the server process in the call message. Procedure ID numbers need only be unique within a server. By convention, procedure ID zero is always the ID of the diagnostic echo routine (RPC_ECHO) provided by the server's RPC interface.

Whenever possible, all binding information should be provided by the stub routine. Sometimes, however, part of the binding information must be supplied by the caller. For example, if the same service is available on more than one node, the node name must be provided by the caller. The diagnostic echo routine, RPC_ECHO, is available on every server on every node. Consequently both the node name and the server name must be supplied by the caller.

5 PARAMETER PASSING

The caller's RPC interface uses the argument list passed to the stub routine to construct the call message. The argument list alone, however, is not sufficient. The RPC interface must also know how to interpret each of the arguments. The interpretation of the argument list is provided by a data structure called the "Parameter Descriptor Block" (PDB) which is supplied by the stub. The PDB contains a four-byte entry for each argument. Each entry describes the parameter's type, its calling mechanism, and its length. The format of a PDB entry is shown below in figure 2.

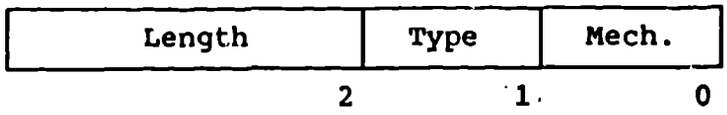

Figure 2
Format of Parameter Descriptor Block Entry

5.1 Calling Mechanism

The first byte of the PDB entry describes how the parameter is passed. The RPC interface recognizes five different mechanisms: pass by value, pass by reference, pass by descriptor, function return, and absent.

The RPC interface automatically detects missing parameters and flags them as such in the call message. The stub routine can also unilaterally flag a parameter absent by declaring it absent in the PDB. For example, a remote procedure may require a "node-name" parameter in order to bind the call to the correct node. Usually, however, the node-name parameter is used only by the RPC interface and not by the procedure itself. In this case, the stub routine would mark the node-name parameter absent in the PDB, and the RPC interface would not incur the overhead of sending an unused parameter across the network.

A "Function Return" PDB entry does not correspond to any parameter in the argument list. Its presence indicates that the procedure is a function and will return a function value.

5.2 Parameter Types

The RPC interface not only needs to know the calling mechanism of a parameter, it also needs to know something about how that parameter is used. The second byte of a PDB entry gives the parameter type. There are seven parameter types: input, output, modified, status,

event, AST address, and AST parameter.

A status parameter is a special case of an output parameter. The way the RPC interface detects and reports errors requires that it know which parameters contain status codes (see section 7). Status parameters are always passed to the remote procedure, even if they were omitted in the caller's argument list or marked absent by the stub.

AST address, AST parameter, and event parameters can be used to signal the end of an asynchronous procedure call. When the remote procedure returns, the caller's RPC interface will set any event flags and declare any AST routines that were in the argument list.

Obviously there can be many different combinations of parameter type and calling mechanism. The RPC interface will detect an invalid combination, such as an output parameter passed by value, and report an error.

6 MESSAGE FORMATS

An RPC call or reply message consists of a message header followed by zero or more "parameter definitions". The message header for a call message is shown below in figure 3.

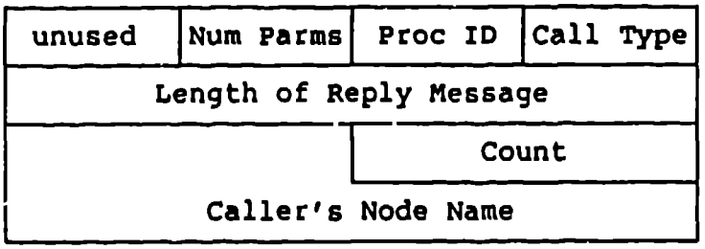

Figure 3
Message Header (Call Message)

The call message header contains the type of call (synchronous,

asynchronous, or no-reply), the ID number of the procedure to call, the number of parameter definitions in the message (including absent parameters and function return values), the expected length of the reply message, and the caller's node name. The server's RPC interface uses the length of the reply message to allocate storage for the reply message. The caller's node name can be used by the server to implement a node-level "access control list" for its remote procedures.

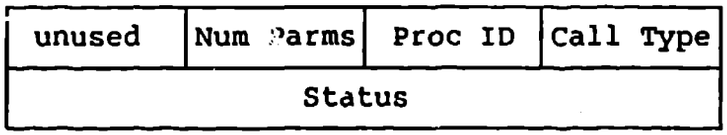

Figure 4
Message Header (Reply Message)

The first four bytes of the reply message (figure 4) have the same function as the first four bytes of the call message. The last four bytes of the reply message contain a status code which is used to report errors detected by the server's RPC interface.

The parameter definitions following the message header contain the information necessary for the server's RPC interface to reconstruct the argument list and for the caller's RPC interface to unpack the output variables. The format of a parameter definition is shown in figure 5.

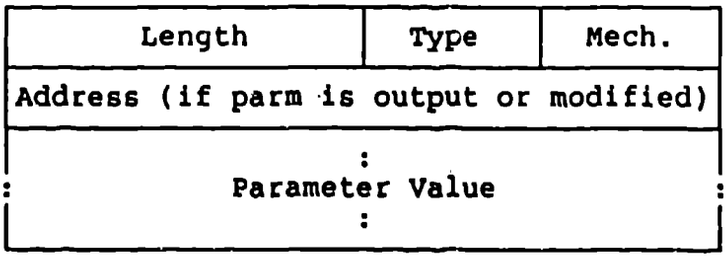

Figure 5
Parameter Definition (One for Each Argument)

The first four bytes of the parameter definition contain the PDB entry for the parameter. If the parameter is an output or modified parameter, then the next four bytes of the parameter definition will contain the address of the parameter (in the caller's address space). This parameter address can not, of course, be used by the server process since the caller and server do not share a common address space. It is present in the RPC message for "preservation", since the original argument list may no longer exist when an asynchronous procedure call returns.

The last part of the parameter definition is the parameter value. Values of input or modified parameters are passed to the server in the call message. Values of output or modified parameters are passed back to the caller in the reply message. Values of event, AST parameter, and AST address parameters are "stored" in the RPC message (passed both ways) so that they will be available to the caller's RPC interface when the procedure returns.

7 ERROR HANDLING

The RPC interface must deal with errors from three sources: 1) errors detected by the remote procedure, 2) errors detected by the RPC interface, and 3) errors detected by DECnet. All errors are handled in a uniform manner.

7.1 Errors Detected By The Remote Procedure

The RPC interface supports two mechanisms for reporting errors which we call the "VMS error standard" and the "VAXELN error standard". In the "VMS error standard", a status code is returned as the procedure's function value. To support the VMS standard, the stub routine need only create a PDB entry with the parameter type "status" and mechanism "function return". In the "VAXELN error standard", one

of the procedure's parameters is an optional status parameter. If the optional status parameter is present, the routine will return a status code in it. If the optional status parameter is absent, and the procedure detects an error, the procedure will raise an exception.

If a parameter is declared in the PDB to be a "status" parameter, the RPC interface will always pass it to the remote procedure -- even if it was omitted from the argument list. When the remote procedure returns, the caller's RPC interface will examine the value of this status parameter. If the status value is not a success code (low-order bit set), and the parameter definition indicates that the parameter is absent, the caller's RPC interface will raise an exception in the caller's process. In this manner errors detected by remote procedures are signaled in the calling process rather than in the server process.

7.2 Errors Detected By The RPC Interface

The RPC interface can detect certain "protocol" errors such as an invalid procedure ID. Errors detected by the RPC interface are handled in the same manner as errors detected by the remote procedure. The RPC interface will search the PDB for a status parameter. If a status parameter is found, and it is present in the argument list, the RPC interface will write a status code into the status parameter and return to the caller. If no status parameter is found, or if the status parameter is absent, the RPC interface will raise an exception.

Another type of error detected by the RPC interface occurs when the remote procedure terminates abnormally (e.g. divide by zero errors, etc.). To catch any such abnormal terminations, the server's RPC interface establishes a condition handler prior to calling the remote procedure. If the remote procedure aborts, the condition handler will store the reason for the abort in the status field of the

reply message header. The caller's RPC interface can then handle the abort like any other error.

7.3 Errors Detected By DECnet

We make the assumption that if DECnet reports a communications error, then it has done everything it can, including retries, to get the message delivered. We therefore will not retry the message ourselves. We also assume that if DECnet succeeded in delivering the message, then the message is correct, so we do not checksum our messages either. Communications errors are treated the same way as any other error, except that when a communications error is detected the RPC interface will shut down the logical link between the caller and the server process.

8 CONSTRUCTING THE STUB ROUTINE

Many RPC systems will automatically generate stubs from information supplied by the compiler or from special "procedure definition files". We did not feel, however, that the standard VAX program development environment would adequately support automatic stub generation.

Since we don't generate stubs automatically, we have tried to make the task of manually creating a stub as easy as possible. In most cases, all that a stub routine has to do is supply the parameter descriptor block (PDB), the procedure binding information, the type of call (synchronous, asynchronous, or no-reply), and a call to RPC_REMOTE_CALL to pass this information on to the RPC interface. The architecture of the VAX procedure call allows the RPC interface to directly access the argument list passed to the stub routine. The stub does not have to pass it explicitly.

Figure 6 illustrates what a stub routine for the RPC_ECHO diagnostic routine might look like.

```
 1  0    MODULE RPC_ECHO_STUB [IDENT('11 SEP 86')];
 2  0
 3  0    EXPORT RPC_ECHO;
 4  0    INCLUDE RPCDEF;                          ( RPC symbol & routine definitions)
 5  0
 6| 0    {
 7| 0    | RPC_ECHO Function Declaration
 8| 0    }
 9  0
10  0    FUNCTION RPC_ECHO (    NODE:           [READONLY]  STRING (<N1>);
11  1                           PROCESS:        [READONLY]  STRING (<N2>);
12  1                           INPUT_STRING:   [READONLY]  STRING (<N3>);
13  1                     VAR OUTPUT_STRING:                STRING (<N4>)
14  1                    ): INTEGER;
15  1
16| 1    {
17| 1    | Define the parameter descriptor block for RPC_ECHO
18| 1    }
19  1
20  1    CONST
21  1      NUM_PARMS = 5;                                  (number of parameters)
22  1
23  1    VAR
24  1       PDB: [READONLY] ARRAY [1..NUM_PARMS] OF _RPCDEF
25  1        := ((RPC_K_ABSENT, RPC_K_INPUT,   0),        (node)
26  1            (RPC_K_ABSENT, RPC_K_INPUT,   0),        (server)
27  1            (RPC_K_DESCR,  RPC_K_INPUT,   0),        (input_string)
28  1            (RPC_K_DESCR,  RPC_K_OUTPUT,  0),        (output_string)
29  1            (RPC_K_FUNC,   RPC_K_STATUS,  4));       (function return)
30  1
31| 1    {
32| 1    | Code for RPC_ECHO
33| 1    }
34  1
35  1    BEGIN
36  1       RPC_ECHO := RPC_REMOTE_CALL (NODE             := NODE,          (node name)
37  1                                    LISTENER         := PROCESS,       (server name)
38  1                                    ROUTINE_ID       := 0,             (RPC_ECHO)
39  1                                    REPLY_FLAG       := RPC_K_SYNCH,   (synchronous call)
40  1                                    NUM_PARMS        := NUM_PARMS,     (number of parms)
41  1                                    TIMEOUT          := 30,            (30 seconds)
42  1                                    PARM_DESC_BLOCK := PDB)            (parm descriptions)
43  1    END (rpc_echo)
44  0    END. (module)

EPASCAL/LIST RPCECHO,LCS_LIB:PASLIB/LIBRARY
```

Figure 6
Sample Stub Routine

Note that the node and server name parameters have been forced "absent" in the PDB. These parameters are only used for binding; the RPC_ECHO routine itself does not reference them. Also note that RPC_ECHO conforms to the "VMS error standard" since a status code is returned as the function value.

9 PERFORMANCE

Figure 7 shows how our RPC system compares with DECnet alone. RPC_ECHO was used to send a block of data over a 10-megabit Ethernet line from a VAX 780 running VMS, to a Micro-VAX II running VAXELN, and back again. The round-trip times are compared with those of another program that just uses DECnet to do the same thing.

Number of Bytes Transferred	Protocol	
	RPC	DECnet
0 Bytes	15.63 ms	13.31 ms
512 Bytes	22.46 ms	18.39 ms
2048 Bytes	41.42 ms	35.03 ms
8192 Bytes	113.81 ms	95.69 ms

Figure 7
Message Echo Times

As can be seen from the table, the RPC interface adds about 20 percent to the DECnet overhead.

10 CONCLUSIONS

The remote procedure call model works well when the communicating processes are in a client/server relationship. The model breaks down, however, when the processes must interact as equals or when broadcast capability is required. Fortunately, the client/server relationship is common in distributed systems such as ours.

We feel that our system has succeeded in merging the simplicity of the RPC model with the power and flexibility of DECnet. The resulting system has been both reliable and easy to use.

DECnet does impose a certain amount of overhead, and our RPC system adds to that overhead. So far this overhead has not been intolerable. In some situations the use of asynchronous procedure calls has lessened the impact of the overhead.

11 ACKNOWLEDGEMENTS

The authors would like to acknowledge the contributions of Stan Brown, Gary Carr, and Jim Harrison for their help and guidance with this project.

REFERENCES

[1] ALMES G.T., BLACK A.P., LAZOWSKA E.D., and NOE J.D., The Eden System: A Technical Review (IEEE Trans. Software Eng. SE-11, January 1985) pp 43-59.

[2] BIRRELL A.D., and NELSON B.J., Implementing Remote Procedure Calls (ACM Trans. Computer Systems, Vol. 2, No. 1, February 1984) pp 39-59.

[3] BLACK A.P., Supporting Distributed Applications: Experience with Eden, (Proceedings, Tenth ACM Symposium on Operating Systems Principles. Operating Systems Review, Vol. 19, No. 5, December 1985) pp 181-193.

[4] ICHBIAH J.D., HELIARD J.C., ROUBINE O., BARNES J.G.P., KRIEG-BRUECKNER B., and WICHMANN B.A. Rational for the Design of the ADA Programming Language, Section II, Tasking (SIGPLAN Notices, Vol. 14, No. 6, Part B, June 1979).

[5] SWINEHART D.C., ZELLWEGER P.T., and HAGMANN R.B. The Structure of Cedar (Proceedings, ACM SIGPLAN '85 Symposium on Language Issues in Programming Environments. SIGPLAN Notices Vol. 20, No. 7, July 1985) pp 230-244.